H

hw3-doc

1 0 https://gitlab02.cs.stonybrook.edu/cse320/hw3-d

Change NUM_FREE_LISTS from 9 to 8, to match the discussion. 2c8b4718
Gene Stark authored 1 day ago

Name	Last commit	Last update
README.md	Change NUM_FREE_LISTS from 9 to 8, to match the discussi...	1 day ago

README.md

Homework 3 Dynamic Memory Allocator - CSE 320 - Fall 2019

Professor Eugene Stark

Due Date: Friday 10/25/2019 @ 11:59pm

We **HIGHLY** suggest that you read this entire document, the book chapter, and examine the base code prior to beginning. If you do not read the entire document before beginning, you may find yourself doing extra work.

> 😱 Start early so that you have an adequate amount of time to test your program!
>
> 😱 The functions `malloc`, `free`, `realloc`, `memalign`, `calloc`, etc., are **NOT ALLOWED** in your implementation. If any of these functions, or any other function with similar functionality is found in your program, you **will receive a ZERO**.

NOTE: In this document, we refer to a word as 2 bytes (16 bits) and a memory row as 4 words (64 bits). We consider a page of memory to be 4096 bytes (4 KB)

Introduction

You must read **Chapter 9.9 Dynamic Memory Allocation Page 839** before starting this assignment. This chapter contains all the theoretical information needed to complete this assignment. Since the textbook has sufficient information about the different design strategies and implementation details of an allocator, this document will not cover this information. Instead, it will refer you to the necessary sections and pages in the textbook.

Takeaways

After completing this assignment, you will have a better understanding of:

- The inner workings of a dynamic memory allocator
- Memory padding and alignment
- Structs and linked lists in C
- errno numbers in C
- Unit testing in C

Overview

You will create an allocator for the x86-64 architecture with the following features:

- Free lists segregated by size class, using first-fit policy within each size class.
- Immediate coalescing of blocks on free with adjacent free blocks.
- Boundary tags to support efficient coalescing.
- Block splitting without creating splinters.
- Allocated blocks aligned to "double memory row" (16-byte) boundaries.
- Free lists maintained using **last in first out (LIFO)** discipline.
- Obfuscation of block footers to detect attempts to free blocks not previously obtained via allocation.

You will implement your own versions of the **malloc**, **realloc**, and **free** functions.

You will use existing Criterion unit tests and write your own to help debug your implementation.

Free List Management Policy

Your allocator **MUST** use the following scheme to manage free blocks: Free blocks will be stored in a fixed array of `NUM_FREE_LISTS` free lists, segregated by size class (see **Chapter 9.9.14 Page 863** for a discussion of segregated free lists). The size classes are based on powers of 2, according to the following: The first free list (at index 0) holds blocks of the minimum size `M` (where `M = 32` for this assignment). The second list (at index 1) holds blocks whose size is in the interval `(M, 2M]`. The third list holds blocks whose size is in the interval `(2M, 4M]`. The fourth list holds blocks whose size is in the interval `(4M, 8M]`, and so on. This pattern continues up to the interval `(64M, 128M]`, and then the last list holds all blocks of size greater than `128M`. Allocation requests will be satisfied by searching the free lists in increasing order of size class. Each individual free list will be organized as a **circular, doubly linked list** (more information below).

Block Placement Policy

When allocating memory in response to an allocation request, use a **segregated fits policy** as follows: First, the smallest size class that is sufficiently large to satisfy the request is determined. The free lists are then searched, starting from the list for the determined size class and continuing in increasing order of size, until a nonempty list is found. The request is then satisfied by the first block in that list that is sufficiently large; *i.e.* a **first-fit policy** (discussed in **Chapter 9.9.7 Page 849**) is applied within each individual free list.

If there is no free list that contains a sufficiently large block, then `sf_mem_grow` should be called to extend the heap by an additional page of memory. After **coalescing** this page (see below) with any free block that immediately precedes it, you should attempt to use the resulting block of memory to satisfy the allocation request; splitting it as usual if it is too large and no splinter would result. If the block of memory is still not large enough, another call to `sf_mem_grow` should be made; continuing to grow the heap until either a large enough block is obtained or the return value from `sf_mem_grow` indicates that there is no more memory.

As discussed in the book, segregated free lists allow the allocator to approximate a best-fit policy, with lower overhead than would be the case if an exact best-fit policy were implemented.

Splitting Blocks & Splinters

Your allocator must split blocks at allocation time to reduce the amount of internal fragmentation. Details about this feature can be found in **Chapter 9.9.8 Page 849**. Note that, as a result of alignment and overhead constraints, the minimum useful block size is 32 bytes. No "splinters" of smaller size than this are ever to be created. If splitting a block to be allocated would result in a splinter, then the block should not be split; rather, the block should be used as-is to satisfy the allocation request (*i.e.*, you will "over-allocate" by issuing a block slightly larger than that required).

> 🤔 Why is the minimum block size 32? As you read more details about the format of a block header, block footer, and alignment requirements, think about how these constrain the minimum block size.

Freeing a Block

When a block is freed, an attempt should first be made to **coalesce** the block with any free block that immediately precedes or follows it in the heap. (See **Chapter 9.9.10 Page 850** for a discussion of the coalescing procedure.) Once the block has been coalesced, it should be inserted at the **front** of the free list for the appropriate size class (based on the size after coalescing). The reason for performing coalescing is to combat the external fragmentation that would otherwise result due to the splitting of blocks upon allocation.

Block Headers & Footers

In **Chapter 9.9.6 Page 847 Figure 9.35**, the a block header is defined as 2 words (32 bits) to hold the block size and allocated bit. In this assignment, the header will be 4 words (i.e. 64 bits or 1 memory row). The header fields will be similar to those in the textbook but you will keep an extra field for recording whether or not the previous block is allocated. Each block will also have a footer, which occupies the last memory row of the block. The footer contains exactly the same information as the header, except that in this assignment the footer will be obfuscated as a security feature (more about this below).

Block Header Format:

```
+---------------------------------------------------------------------+---------+---------+ <- header
|                                    block_size                       |  alloc  |prv alloc|
|                                (2 LSB's implicitly 0)               |   (1)   |  (0/1)  |
|                                       64 bits                       |  1 bit  |  1 bit  |
+---------------------------------------------------------------------+---------+---------+ <- (aligned)
```

- The `block_size` field gives the number of bytes for the **entire** block (including header/footer, payload, and padding). It occupies the entire 64 bits of the block header or footer, except that the two least-significant bits of the block size, which would normally always be zero due to alignment requirements, are used to store the allocation status (free/allocated) of that block and of the immediately preceding block in the heap. This means that these two bits have to be masked when retrieving the block size from the header and when the block size is stored in the header the previously existing values of these two bits have to be preserved.
- The `alloc` bit is a boolean. It is 1 if the block is allocated and 0 if it is free.
- The `prev_alloc` bit is also a boolean. It is 1 if the **immediately preceding** block in the heap is allocated and 0 if it is not.

> 🤔 Here is an example of determining the block size required to satisfy a particular requested payload size. Suppose the requested size is 25 bytes. An additional 16 bytes will be required to store the block header and footer, which must always be present. That means a block of at least 41 bytes must be used, however due to alignment requirements this has to be rounded up to the next multiple of 16, which is 48. As a result, there will be 7 bytes of "padding" at the end of the payload area, which contributes to internal fragmentation. Besides the header and footer, it is also necessary to store next and previous links for the freelist when the block is free. These will take an additional 16 bytes of space, however when the block is free there is no payload so the payload area can be used to store this information, assuming that the payload area is big enough in the first place. But the payload area is 32 bytes (25 bytes plus 7 bytes of padding), which is certainly bigger than 24 bytes, so a block of total size 48 will be fine. Note that if a block is smaller than 32 bytes, there would not be enough space to store the header, footer, and freelist links when the block is free. This is why the minimum block size is 32 bytes.

Getting Started

Fetch and merge the base code for `hw3` as described in `hw0` from the following link: https://gitlab02.cs.stonybrook.edu/cse320/hw3

Remember to use the `--strategy-option=theirs` flag with the `git merge` command as described in the `hw1` doc to avoid merge conflicts in the Gitlab CI file.

Directory Structure

```
.
├── .gitlab-ci.yml
└── hw3
    ├── include
    │   ├── debug.h
    │   └── sfmm.h
    ├── lib
    │   └── sfutil.o
    ├── Makefile
    ├── src
    │   ├── main.c
    │   └── sfmm.c
    └── tests
        └── sfmm_tests.c
```

The `lib` folder contains the object file for the `sfutil` library. This library provides you with several functions to aid you with the implementation of your allocator. **Do NOT delete this file as it is an essential part of your homework assignment.**

The provided `Makefile` creates object files from the `.c` files in the `src` directory, places the object files inside the `build` directory, and then links the object files together, including `lib/sfutil.o`, to make executables that are stored to the `bin` directory.

Note: `make clean` will not delete `sfutil.o` or the `lib` folder, but it will delete all other contained `.o` files.

The `sfmm.h` header file contains function prototypes and defines the format of the various data structures that you are to use.

> 😱 **DO NOT modify `sfmm.h` or the Makefile.** Both will be replaced when we run tests for grading. If you wish to add things to a header file, please create a new header file in the `include` folder

All functions for your allocator (`sf_malloc`, `sf_realloc`, and `sf_free`) **must be** implemented in `src/sfmm.c`.

The program in `src/main.c` contains a basic example of using the initialization and allocation functions together. Running `make` will create a `sfmm` executable in the `bin` directory. This can be run using the command `bin/sfmm`.

> Any functions other than `sf_malloc`, `sf_free`, and `sf_realloc` **WILL NOT** be graded.

Allocation Functions

You will implement the following three functions in the file `src/sfmm.c`. The file `include/sfmm.h` contains the prototypes and documentation found here.

Standard C library functions set `errno` when there is an error. To avoid conflicts with these functions, your allocation functions will set `sf_errno`, a variable declared as `extern` in `sfmm.h`.

```
/*
 * This is your implementation of sf_malloc. It acquires uninitialized memory that
 * is aligned and padded properly for the underlying system.
 *
 * @param size The number of bytes requested to be allocated.
 *
 * @return If size is 0, then NULL is returned without setting sf_errno.
 * If size is nonzero, then if the allocation is successful a pointer to a valid region of
 * memory of the requested size is returned.  If the allocation is not successful, then
 * NULL is returned and sf_errno is set to ENOMEM.
 */
void *sf_malloc(size_t size);

/*
 * Resizes the memory pointed to by ptr to size bytes.
 *
 * @param ptr Address of the memory region to resize.
 * @param size The minimum size to resize the memory to.
 *
 * @return If successful, the pointer to a valid region of memory is
 * returned, else NULL is returned and sf_errno is set appropriately.
 *
 *   If sf_realloc is called with an invalid pointer sf_errno should be set to EINVAL.
 *   If there is no memory available sf_realloc should set sf_errno to ENOMEM.
 *
 * If sf_realloc is called with a valid pointer and a size of 0 it should free
 * the allocated block and return NULL without setting sf_errno.
 */
void* sf_realloc(void *ptr, size_t size);

/*
 * Marks a dynamically allocated region as no longer in use.
 * Adds the newly freed block to the free list.
 *
 * @param ptr Address of memory returned by the function sf_malloc.
 *
```

```
 * If ptr is invalid, the function calls abort() to exit the program.
 */
void sf_free(void *ptr);
```

😱 Make sure these functions have these exact names and arguments. They must also appear in the correct file. If you do not name the functions correctly with the correct arguments, your program will not compile when we test it. **YOU WILL GET A ZERO**

Initialization Functions

In the `build` directory, we have provided you with the `sfutil.o` object file. When linked with your program, this object file allows you to access the `sfutil` library, which contains the following functions:

This library contains the following functions:

```
/*
 * Any program using the sfmm library must call this function ONCE
 * before issuing any allocation requests. This function DOES NOT
 * allocate any space to your allocator.
 */
void sf_mem_init();

/*
 * Any program using the sfmm library must call this function ONCE
 * after all allocation requests are complete. If implemented cleanly,
 * your program should have no memory leaks in valgrind after this function
 * is called.
 */
void sf_mem_fini();

/*
 * This function increases the size of your heap by adding one page of
 * memory to the end.
 *
 * @return On success, this function returns a pointer to the start of the
 * additional page, which is the same as the value that would have been returned
 * by sf_mem_end() before the size increase.  On error, NULL is returned
 * and sf_errno is set to ENOMEM.
 */
void *sf_mem_grow();

/* The size of a page of memory returned by sf_mem_grow(). */
#define PAGE_SZ 4096

/*
 * @return The starting address of the heap for your allocator.
 */
void *sf_mem_start();

/*
 * @return The ending address of the heap for your allocator.
 */
void *sf_mem_end();

/*
 * @return The "magic number" used to obfuscate footer contents to make it
 * difficult to free a block without having first succesfully malloc'ed one.
 */
uint64_t sf_magic();
```

The function `sf_mem_init` **MUST** be used to initialize memory. It is to be called once **BEFORE** using any of the other functions from `sfutil.o`. The function `sf_mem_grow` is to be invoked by `sf_malloc`, at the time of the first allocation request to set up the heap

prologue and epilogue and obtain an initial free block, and on subsequent allocations when a large enough block to satisfy the request is not found.

> 😱 As these functions are provided in a pre-built .o file, the source is not available to you. You will not be able to debug these using gdb. You must treat them as black boxes.

sf_mem_grow

For this assignment, your implementation **MUST ONLY** use `sf_mem_grow` to extend the heap. **DO NOT** use any system calls such as **brk** or **sbrk** to do this.

Function `sf_mem_grow` returns memory to your allocator in pages. Each page is 4096 bytes (4 KB) and there are a limited, small number of pages available (the actual number may vary, so do not hard-code any particular limit into your program). Each call to `sf_mem_grow` extends the heap by one page and returns a pointer to the new page (this will be the same pointer as would have been obtained from `sf_mem_end` before the call to `sf_mem_grow`.

The `sf_mem_grow` function also keeps track of the starting and ending addresses of the heap for you. You can get these addresses through the `sf_mem_start` and `sf_mem_end` functions.

> 😄 A real allocator would typically use the **brk**/**sbrk** system calls calls for small memory allocations and the **mmap**/**munmap** system calls for large allocations. To allow your program to use other functions provided by glibc, which rely on glibc's allocator (*i.e.* `malloc`), we have provided `sf_mem_grow` as a safe wrapper around **sbrk**. This makes it so your heap and the one managed by glibc do not interfere with each other.

Implementation Details

Memory Row Size

The table below lists the sizes of data types (following Intel standard terminlogy) on x86-64 Linux Mint:

C declaration	Data type	x86-64 Size (Bytes)
char	Byte	1
short	Word	2
int	Double word	4
long int	Quadword	8
unsigned long	Quadword	8
pointer	Quadword	8
float	Single precision	4
double	Double precision	8
long double	Extended precision	16

> 🤓 You can find these sizes yourself using the sizeof operator. For example, `printf("%lu\n", sizeof(int))` prints 4.

In this assignment we will assume that each "memory row" is 8 bytes (64 bits) in size. All pointers returned by `sf_malloc` are to be "double memory row" aligned; that is, they will be addresses that are multiples of 16. This will permit such pointers to be used to store values of the largest data type (`long double`) supported by the hardware.

Block Header & Footer Fields

The various header and footer formats are specified in `include/sfmm.h`:

```
                            Format of an allocated memory block
    +-------------------------------------------------------------------------------------+
    |                                      64-bits wide                                   |
    +-------------------------------------------------------------------------------------+

    +-----------------------------------------------------------------+---------+---------+ <- header
    |                                    block_size                   |  alloc  |prv alloc|
    |                                (2 LSB's implicitly 0)           |   (1)   |  (0/1)  |
    |                                       64 bits                   |  1 bit  |  1 bit  |
    +-----------------------------------------------------------------+---------+---------+ <- (aligned)
    |                                                                                       |
    |                                  Payload and Padding                                  |
    |                                    (N Memory Rows)                                    |
    |                                                                                       |
    |                                                                                       |
    +---------------------------------------+-------------------------+---------+---------+ <- footer
    |                                    block_size                   |  alloc  |prv alloc|
    |                                (2 LSB's implicitly 0)           |   (1)   |  (0/1)  |
    |                                       64 bits                   |  1 bit  |  1 bit  |
    +-----------------------------------------------------------------+---------+---------+

    NOTE: Footer contents must always be identical to header contents XOR'ed with sf_magic().
```

```
                                Format of a free memory block
    +-----------------------------------------------------------------+---------+---------+ <- header
    |                                    block_size                   |  alloc  |prv alloc|
    |                                (2 LSB's implicitly 0)           |   (0)   |  (0/1)  |
    |                                       64 bits                   |  1 bit  |  1 bit  |
    +---------------------------------------+-------------------------+---------+---------+ <- (aligned)
    |                                                                                       |
    |                                Pointer to next free block                             |
    |                                                                                       |
    +-------------------------------------------------------------------------------------+
    |                                                                                       |
    |                               Pointer to previous free block                          |
    |                                                                                       |
    +-------------------------------------------------------------------------------------+
    |                                                                                       |
    |                                          Unused                                       |
    |                                     (N Memory Rows)                                   |
    |                                                                                       |
    |                                                                                       |
    +---------------------------------------+-------------------------+---------+---------+ <- footer
    |                                    block_size                   |  alloc  |prv alloc|
    |                                (2 LSB's implicitly 0)           |   (0)   |  (0/1)  |
    |                                       64 bits                   |  1 bit  |  1 bit  |
    +-----------------------------------------------------------------+---------+---------+

    NOTE: Footer contents must always be identical to header contents XOR'ed with sf_magic().
```

The `sfmm.h` header file contains C structure definitions corresponding to the above diagrams:

```
#define PREV_BLOCK_ALLOCATED  0x1
#define THIS_BLOCK_ALLOCATED  0x2
#define BLOCK_SIZE_MASK 0xfffffffc

typedef size_t sf_header;
typedef size_t sf_footer;

/* Structure of a block. */
```

```
typedef struct sf_block {
    sf_footer prev_footer;  // NOTE: This actually belongs to the *previous* block.
    sf_header header;
    union {
        /* A free block contains links to other blocks in a free list. */
        struct {
            struct sf_block *next;
            struct sf_block *prev;
        } links;
        /* An allocated block contains a payload (aligned), starting here. */
        char payload[0];   // Length varies according to block size.
    } body;
} sf_block;
```

For `sf_block`, the `body` field is a `union`, which has been used to emphasize the difference between the information contained in a free block and that contained in an allocated block. If the block is free, then its `body` has a `links` field, which is a `struct` containing `next` and `prev` pointers. If the block is allocated, then its `body` does not have a `links` field, but rather has a `payload`, which starts at the same address that the `links` field would have started if the block were free. The size of the `payload` is obviously not zero, but as it is variable and only determined at run time, the `payload` field has been declared to be an array of length 0 just to enable the use of `bp->body.payload` to obtain a pointer to the payload area, if `bp` is a pointer to `sf_block`.

> 👍 You can use casts to convert a generic pointer value to one of type `sf_block` or `sf_header`, in order to make use of the above structure definitions to easily access the various fields.

Logically, the footer contents of a block are always to be maintained identical to the header contents (and this is how they are shown in the diagrams above). However, as a security feature to help catch freeing of invalid blocks, what will actually be stored in the footer field is the bitwise XOR (C operator `^`) of the header contents with a "magic number" that is obtained by calling `sf_magic()`. The magic number is set upon initialization of the heap and it will be different from one execution to the next. This is intended to make it difficult to free a block (either inadvertently or maliciously) that was not previously obtained from `malloc()`.

> 😱 Note that the `prev_footer` field is actually part of the **previous** block in the heap. In order to initialize an `sf_block` pointer to correctly access the fields of a block, it is necessary to compute the address of the footer of the immediately preceding block in the heap and then cast that address to type `sf_block *`. The footer of a particular block can be obtained by first getting an `sf_block *` pointer for that block and then using the contained information (*i.e.* the block size) to obtain the `prev_footer` field of the **next** block in the heap. The `sf_block` structure has been specified this way so as to permit it to be defined with a fixed size, even though the payload size is unknown and will vary.

Free List Heads

In the file `include/sfmm.h`, you will see the following declaration:

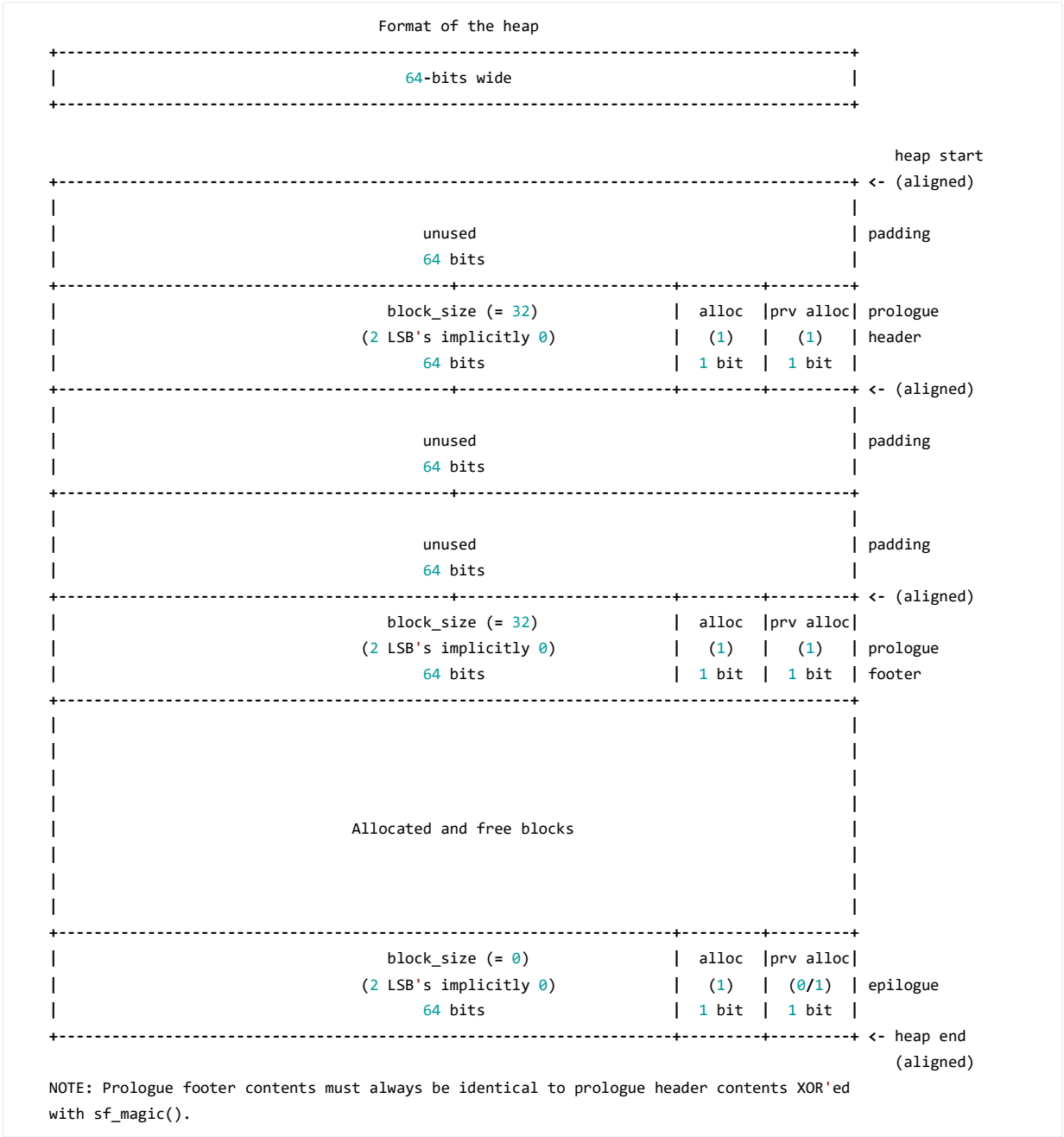
```
#define NUM_FREE_LISTS 8
struct sf_block sf_free_list_heads[NUM_FREE_LISTS];
```

The array `sf_free_list_heads` contains the heads of the free lists, which are maintained as **circular, doubly linked lists**. Each node in a free list contains a `next` pointer that points to the next node in the list, and a `prev` pointer that points the previous node. For each index `i` with `0 <= i < NUM_FREE_LISTS` the variable `sf_free_list_head[i]` is a dummy, "sentinel" node, which is used to connect the beginning and the end of the list at index `i`. This sentinel node is always present and (aside from its `next` and `free` pointers) does **not** contain any other data. If the list is empty, then the fields `sf_freelist_heads[i].body.links.next` and `sf_freelist_heads[i].body.links.prev` both contain `&sf_freelist_heads[i]` (*i.e.* the sentinel node points back to itself). If the list is nonempty, then `sf_freelist_heads[i].body.links.next` points to the first node in the list and `sf_freelist_heads[i].body.links.prev` points to the last node in the list. Inserting into and deleting from a circular doubly linked list is done in the usual way, except that, owing to the use of the sentinel, there are no edge cases for inserting or removing at the beginning or the end of the list. If you need a further introduction to this data structure, you can readily find information on it by googling ("circular doubly linked lists with sentinel").

> 😱 You **MUST** use the `sf_free_list_heads` array for the heads of your free lists and you **MUST** maintain these lists as circular, doubly linked lists. The helper functions discussed later, as well as the unit tests, will assume that you have done this when accessing your free lists.

Overall Structure of the Heap: Prologue and Epilogue

Your heap should use a prologue and epilogue (as described in the book, **page 855**) to arrange for the proper block alignment and to avoid edge cases when coalescing blocks. The overall organization of the heap is as shown below:



The "prologue" consists of padding (to achieve the required alignment) and an allocated block with just a header and a footer and a minimum-size payload area, which is unused.

The "epilogue" consists only of an allocated header, with block size set to 0. The prologue and epilogue are never freed. When the heap is extended, a new epilogue is created at the end of the newly added region and the old epilogue becomes the header of the new block. This is as described in the book.

```
/* Structure of the prologue. */
typedef struct sf_prologue {
    sf_footer padding1;
    sf_header header;
    void *unused1;
    void *unused2;
```

```
    sf_footer footer;
} sf_prologue;

/* Structure of the epilogue. */
typedef struct sf_epilogue {
    sf_header header;
} sf_epilogue;
```

As your heap is initially empty, at the time of the first call to `sf_malloc` you will need to make one call to `sf_mem_grow` to obtain a page of memory within which to set up the prologue and initial epilogue. The remainder of the memory in this first page should then be inserted into the free list as a single block.

Notes on sf_malloc

When implementing your `sf_malloc` function, first determine if the request size is 0. If so, then return `NULL` without setting `sf_errno`. If the request size is non-zero, then you should determine the size of the block to be allocated by adding the header size and the size of any necessary padding to reach a size that is a multiple of 16 to maintain proper alignment. Remember also that the block has to be big enough to store the footer (which will always be present), as well as the `next` and `prev` pointers when the block is free. As the `next` and `prev` pointers are not present in an allocated block this space can (and should) be overlapped with the payload area. The above constraints lead to a minimum block size of 32 bytes, so you should not attempt to allocate any block smaller than this. After having determined the required block size, you should determine the smallest free list that would be able to satisfy a request of that size. Search that free list from the beginning until the first sufficiently large block is found. If there is no such block, continue with the next larger size class. If a big enough block is found, then after splitting it (if it will not leave a splinter), you should insert the remainder part back into the appropriate freelist. When splitting a block, the "lower part" should be used to satisfy the allocation request and the "upper part" should become the remainder. If there is no block big enough in any freelist, then you must use `sf_mem_grow` to request more memory. (For requests larger than a page, more than one such call might be required). If your allocator ultimately cannot satisfy the request, your `sf_malloc` function must set `sf_errno` to `ENOMEM` and return `NULL`.

Notes on sf_mem_grow

After each call to `sf_mem_grow`, you must attempt to coalesce the newly allocated page with any free block immediately preceding it, in order to build blocks larger than one page. Insert the new block at the beginning of the appropriate free list.

Note: Do not coalesce with the prologue or epilogue, or past the beginning or end of the heap.

Notes on sf_free

When implementing `sf_free`, you must first verify that the pointer being passed to your function belongs to an allocated block. This can be done by examining the fields in the block header and footer. In this assignment, we will consider the following cases to be invalid pointers:

- The pointer is `NULL`.
- The `allocated` bit in the header is 0.
- The header of the block is before the end of the prologue, or the footer of the block is after the beginning of the epilogue.
- The `block_size` field is less than the minimum block size of 32 bytes.
- The `prev_alloc` field is 0, indicating that the previous block is free, but the `alloc` field of the previous block header is not 0.
- The bitwise XOR of the footer contents and the value returned by `sf_magic()` does not equal the header contents.

If an invalid pointer is passed to your function, you must call `abort` to exit the program. Use the man page for the `abort` function to learn more about this.

After confirming that a valid pointer was given, you must free the block. First, the block must be coalesced with any adjacent free block. Then, determine the size class appropriate for the (now-coalesced) block and inserting the block at the beginning of the free list for that size class. Note that blocks in a free list must **not** be marked as allocated.

Notes on sf_realloc

When implementing your `sf_realloc` function, you must first verify that the pointer and size parameters passed to your function are valid. The criteria for pointer validity are the same as those described in the 'Notes on sf_free' section above. If the pointer is valid but the size parameter is 0, free the block and return `NULL`.

After verifying both parameters, consider the cases described below. Note that in some cases, `sf_realloc` is more complicated than

calling `sf_malloc` to allocate more memory, `memcpy` to move the old memory to the new memory, and `sf_free` to free the old memory.

Reallocating to a Larger Size

When reallocating to a larger size, always follow these three steps:

1. Call `sf_malloc` to obtain a larger block.
2. Call `memcpy` to copy the data in the block given by the client to the block returned by `sf_malloc`.
3. Call `sf_free` on the block given by the client (coalescing if necessary).
4. Return the block given to you by `sf_malloc` to the client.

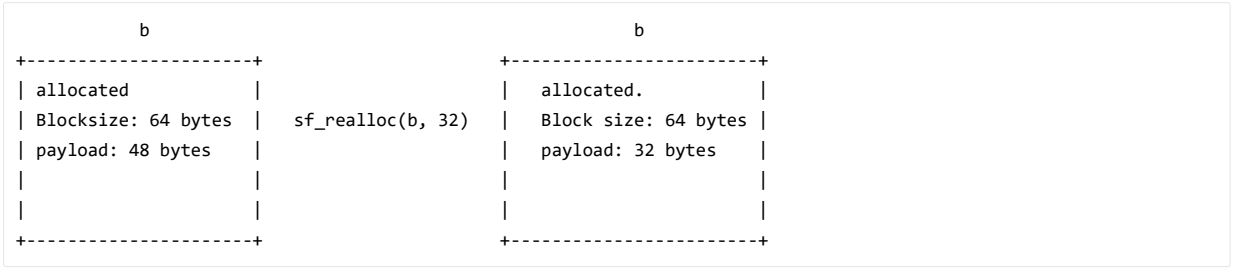
If `sf_malloc` returns `NULL`, `sf_realloc` must also return `NULL`. Note that you do not need to set `sf_errno` in `sf_realloc` because `sf_malloc` should take care of this.

Reallocating to a Smaller Size

When reallocating to a smaller size, your allocator must use the block that was passed by the caller. You must attempt to split the returned block. There are two cases for splitting:

- Splitting the returned block results in a splinter. In this case, do not split the block. Leave the splinter in the block, update the header field if necessary, and return the same block back to the caller.

Example:

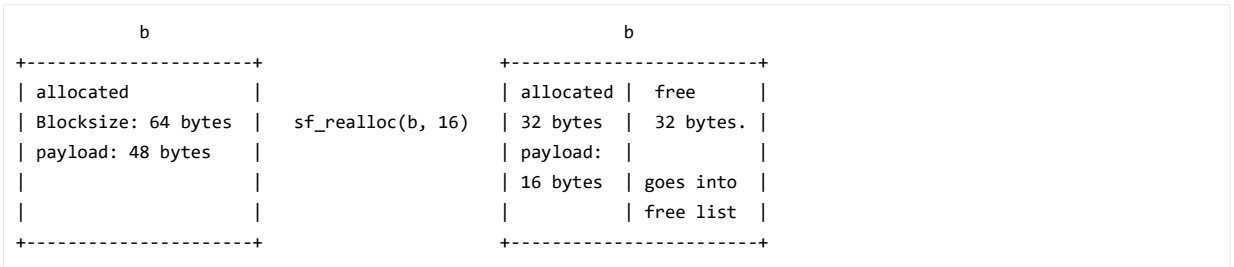


In the example above, splitting the block would have caused a 16-byte splinter. Therefore, the block is not split.

- The block can be split without creating a splinter. In this case, split the block and update the block size fields in both headers. Free the remaining block by inserting it into the appropriate free list (after coalescing, if possible). Return a pointer to the payload of the now-smaller block to the caller.

Note that in both of these sub-cases, you return a pointer to the same block that was given to you.

Example:



Helper Functions

The `sfutil` library additionally contains the following helper functions, which should be self explanatory. They all output to `stderr`.

```
void sf_show_block(sf_block *bp);
void sf_show_blocks();
```

```
void sf_show_free_list(int index);
void sf_show_free_lists();
void sf_show_heap();
```

We have provided these functions to help you visualize your free lists and allocated blocks.

Things to Remember

- Make sure that memory returned to the client is aligned and padded correctly for the system we use (64-bit Linux Mint).
- We will not grade using Valgrind. However, you are encouraged to use it to detect alignment errors.

Unit Testing

For this assignment, we will use Criterion to test your allocator. We have provided a basic set of test cases and you will have to write your own as well.

You will use the Criterion framework alongside the provided helper functions to ensure your allocator works exactly as specified.

In the `tests/sfmm_tests.c` file, there are ten unit test examples. These tests check for the correctness of `sf_malloc`, `sf_realloc`, and `sf_free`. We provide some basic assertions, but by no means are they exhaustive. It is your job to ensure that your header/footer bits are set correctly and that blocks are allocated/freed as specified.

Compiling and Running Tests

When you compile your program with `make`, a `sfmm_tests` executable will be created in the `bin` folder alongside the `main` executable. This can be run with `bin/sfmm_tests`. To obtain more information about each test run, you can use the verbose print option: `bin/sfmm_tests --verbose=0`.

Writing Criterion Tests

The first test `malloc_an_Integer_check_freelist` tests `sf_malloc`. It allocates space for an integer and assigns a value to that space. It then runs an assertion to make sure that the space returned by `sf_malloc` was properly assigned.

```
cr_assert(*x == 4, "sf_malloc failed to give proper space for an int!");
```

The string after the assertion only gets printed to the screen if the assertion failed (i.e. `*x != 4`). However, if there is a problem before the assertion, such as a SEGFAULT, the unit test will print the error to the screen and continue to run the rest of the unit tests.

For this assignment **you must write 5 additional unit tests which test new functionality and add them to `sfmm_tests.c` below the following comment:**

```
//############################################
//STUDENT UNIT TESTS SHOULD BE WRITTEN BELOW
//DO NOT DELETE THESE COMMENTS
//############################################
```

> For additional information on Criterion library, take a look at the official documentation located here! This documentation is VERY GOOD.

Hand-in instructions

Make sure your directory tree looks like this and that your homework compiles.

```
.
├── .gitlab-ci.yml
└── hw3
    ├── include
    │   ├── debug.h
```

```
    │   └── sfmm.h
    ├── lib
    │   └── sfutil.o
    ├── Makefile
    ├── src
    │   ├── main.c
    │   └── sfmm.c
    └── tests
        └── sfmm_tests.c
```

This homework's tag is: `hw3`

```
$ git submit hw3
```

A Word to the Wise

This program will be very difficult to get working unless you are extremely disciplined about your coding style. Think carefully about how to modularize your code in a way that makes it easier to understand and avoid mistakes. Verbose, repetitive code is error-prone and **evil!** When writing your program try to comment as much as possible. Format the code consistently. It is much easier for your TA and the professor to help you if we can quickly figure out what your code does.